DSTO AUSTRALIA

ELECTRONICS RESEARCH LABORATORY

Information Technology Division

TECHNICAL REPORT
ERL-0526-TR

RAPID PROTOTYPING : FOR WANT OF BETTER WORDS

by

Stephen P. Jones

SUMMARY

Rapid prototyping is widely promoted as being an effective technique to assist the development of complex systems. While the phrase is commonly used and discussed within the software system development community, the interpretation is somewhat diverse and the technique itself has not been widely adopted in practice. This paper reviews the recent literature on the subject of 'rapid prototyping' and in highlighting key texts, aims to provide a definition of the terminology and tries to put prototyping into context as a necessary and integral component of the software system development life cycle. The paper discusses the incompatibility of prototyping and the traditional life cycle model and proposes an alternative life cycle model which incorporates prototyping as an integral activity.

SEPT 1990

APPROVED FOR PUBLIC RELEASE

POSTAL ADDRESS: Director, Electronics Research Laboratory,PO Box 1600, Salisbury, South Australia, 5108.

CONTENTS

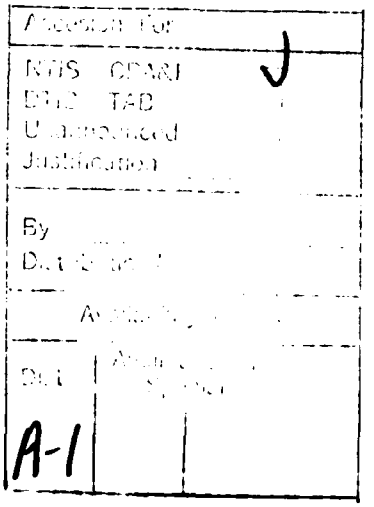
LIST OF FIGURES

1 Introduction

The major proportion of systems being developed today are highly dependent on software in order to properly function. The software content of these systems is larger and more complex than ever before and the term 'software system engineering' is often used to describe the process of management and development of such software intensive projects. Software system engineering is but one part of the total system engineering process, but unlike system engineering, it is still very immature both in terms of the development process itself and the definition of terminology used throughout the process. This paper considers the use of the term 'rapid prototype' within the context of the software system engineering process.

The phrase 'rapid prototyping' is becoming more commonly used within the software system development community to describe the process of generation of pre-production items that resemble the end products to varying degrees of representation and whose intended uses are extremely diverse. One particular rapid prototype may exhibit only superficial resemblance to the end product while another may display a high degree of equivalence to the end product in every way. The absence of a common definition or framework for discussion leads to widely differing interpretations of the terminology and serves to promote confusion between current and prospective users of that terminology.

This paper surveys the field of rapid prototyping with a view to providing an insight into what is meant, for want of better words, by the phrase 'rapid prototyping'. The paper presents both a definition and a framework around which to discuss the topic of rapid prototyping, gleaned from research of numerous articles published in the field, and discusses the need for rapid prototyping throughout the software system development life cycle within the context of this framework. The paper addresses the impact that rapid prototyping has on the traditional software system development life cycle and discusses an alternative development life cycle approach which accommodates rapid prototyping. Finally the paper discusses different approaches to prototyping and highlights the use of current and possible future tools to assist with the exercise.



THIS PAGE INTENTIONALLY LEFT BLANK

2 Surveying the Literature

A survey of the available literature dedicated to the topic of 'rapid prototyping' highlights the varied interpretation and scope applied to the use of the terminology throughout the software system engineering field. This diversity of use of the terminology has prompted numerous papers proposing possible definitions and/or frameworks around which to discuss the topic.

2.1 An Analysis of the Terminology

The word 'prototype' is derived from the Greek 'protos' meaning first and is used outside the software system engineering field to describe the original item in relation to any further copy, imitation, representation or improved product. Gregory [1] uses the traditional engineering definition when he describes a prototype as 'containing all of the final products functionality and to all intent and purpose as being a hand crafted version of the final production model'. This highlights the gross misuse of the term 'prototype' in the software system engineering field, where it is commonly used to define the generation of an item which may only bear a partial resemblance to the final product.

The adjective 'rapid' over-emphasises the short development time normally expected for the production of a prototype, to the extent that it suggests that either a magical solution is employed or free license is given to 'hack' a solution. The word does not convey the relatively short time expected for the production of a prototype as compared to the very long development lead times associated with the production item.

While the terminology is less than perfect, it has been widely used within the software system engineering community for long enough to have gained a foothold and is unlikely to be replaced. In accepting this fact, it is vital that the popular misconceptions accompanying the terminology are removed and a common definition and framework around which to discuss the terminology is identified.

2.2 A Possible Definition

Gregory [1] and Weiser [2] both found the need to use alternative terminology in order to provide a more semantically correct definition for use in the software system engineering field. The terms 'mock-up' and 'scale-model' were used respectively, to convey the experimental nature and non equivalence of the generated item when compared to the final product. The terminology was chosen to suggest the early introduction into the production life cycle along with limited but natural interaction with the intended environment.

Weiser's [2] definition of the terminology, which follows, provides a suitable basis for discussion, but it will become evident that this definition does not totally cover all aspects of today's use of the terminology.

'A prototype of a system is a model of that system which sacrifices accuracy in some areas for a quick check of the systems functionality. A prototype is one kind of scale-model, a model accurate in some ways but inaccurate in others'.

This definition includes a less emphatic reference to speed of production, and makes redundant the adjective in 'rapid prototyping'. The unqualified term 'prototype' is therefore used throughout the remainder of the discussion.

2.3 A Framework for Discussion

Given the diversity of use of the term 'prototype' and the increasing interest in adopting the technique to assist software system development, the need for a framework in which to discuss the topic is crucial.

2.3.1 Classes of Prototypes

Floyd [3] and Law [4] realised the need not only to provide an overall definition of the term 'prototype' but moreover the need to provide a structure in which to discuss classes of prototype. Floyd [3] classified prototypes as being one of either **exploratory**, **experimental** or **evolutionary**. Law [4] extended this classification to include **performance** and **organisational** prototypes. The value of such classification is not only the provision of a framework and suitable terminology around which to discuss prototyping but also to emphasise the scope of prototyping and highlight where a particular class of prototyping is applicable within the software system development life cycle.

The five classes of prototype defined by Law [4] are summarised as follows:

- **Exploratory** prototypes provide the early focus for discussion during the requirements elicitation and verification phases.
- **Experimental** prototypes provide the platform on which to evaluate the proposed software system design.
- **Evolutionary** prototypes are concerned with the gradual adaptation of operational systems to cater for newly identified or changing requirements, the previous operational system serving as the prototype for the new development. This class is the most controversial and is often mistakenly equated to incremental development. It could be argued that this class lies outside the scope of prototyping as we have previously defined it. This does however raise the question as to whether or not a prototype should be a throw away item or a product of an evolutionary development life cycle.
- **Performance** prototypes are concerned with confirmation that the system will meet its stated requirements in terms of performance, response and system loading. This could be considered to be a special case of the experimental prototype class.
- **Organisational** prototypes are concerned with exercising the system in the end user environment, in order to confirm completeness of the solution and compatibility of the existing work practices. This may highlight the need for new manual procedures to be adopted, new job descriptions to be written or training to be provided. This again may be considered a special case of the experimental prototype class although there is a distinct difference in its purpose.

2.3.2 An Alternative Prototype Classification

A useful and preferred alternative approach to classification, put forward by Mayhew and Dearnley [5], considers the four components of a prototype system; the **Prototyper**, the **User**, the **Software** and the **Hardware** and how each component interacts with each other. This prototype classification is assisted by means of their 'PUSH' pyramid, illustrated in Figure 1, where each of the vertices represents a component and each connecting edge their respective interaction.

Mayhew and Dearnley [5] use their 'PUSH' pyramid to interpret Law's [4] five classes of prototype through analysis of how they relate to the interacting components. The result is the derivation of an alternative classification which introduces the flexibility to consider classes of prototype not just by their applicability to particular stages of the development life cycle but also by the importance attributed to the interaction of the key components of any prototype system.

Mayhew and Dearnley's classification [5], introduces a prototype category for each pair of interacting components represented on their 'PUSH' pyramid. These six categories are summarised as follows:

- **Exploratory**, used to establish system requirements, concentrating on the communication between the system and the user.
- **Experimental**, used to verify aspects of the software design.
- **Performance**, used to establish whether or not the target system will handle the anticipated load.
- **Hardware**, used to establish the suitability of the chosen hardware.
- **Ergonomic**, used to assess the physical acceptability of the system.
- **Functional**, used to verify the completeness of aspects of the system.

2.3.3 The Preferred Prototype Classification

Mayhew and Dearnley [5] further group their six categories to form three distinct classes of prototype as shown:

- **Exploratory**.

Exploratory.

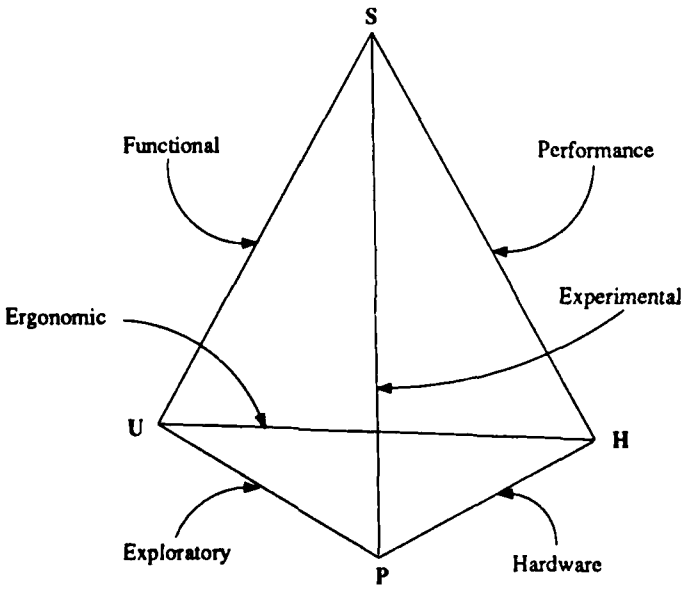


Figure 1: Mayhew and Dearnley's 'PUSH' Pyramid.

- **Experimental.**

  Experimental.
  Performance.
  Hardware.

- **Organisational.**

  Ergonomic.
  Functional.

Although based on Law's[4] classification, the emphasis of Mayhew and Dearnley's [5] classification is centred around both the primary intent of the prototype and the key participants of the prototyping exercise. Mayhew and Dearnley [5] point out that, 'any one prototype may contain elements of many categories outlined above; however it is important to be able to recognise and separate each one of these individual intentions'.

Figure 2 provides a slightly modified representation of Mayhew and Dearnley's [5] alternative classification of prototypes.

For the sake of clarity, Figure 2 has an additional column depicting 'sub-class', which has been added to the representation, and the term 'design' introduced at the sub-class level to replace the repeated use of 'experimental'.

It should be noted that this alternative classification omits Law's [4] class of evolutionary prototype, but it is accepted that prototypes themselves may be evolutionary and an exploratory prototype may evolve into an experimental prototype. This again raises the question as to whether a prototype should further evolve into the end product or be treated as a separate throw away item.

This alternative classification serves to assist the prototyper in focusing his attention on the intent of the prototype and the role of the interacting components. This is important in defining the prototype itself and also the objectives of the prototype evaluation exercise. This classification, along with the previous definition of the term 'prototype', will form the framework for further discussion.

Class	Sub-Class	Interaction	Comment
Exploratory	N/A	Prototyper/User	Requirements elicitation and validation
Experimental	Design	Prototyper/Software	Testing the software design
	Performance	Software/Hardware	Will the proposed system cope?
	Hardware	Prototyper/Hardware	Is the chosen hardware suitable?
Organisational	Ergonomic	User/Hardware	Hardware set-up considerations
	Functional	User/Software	Complete system suitability

Figure 2: Mayhew and Dearnley's Alternative Classification.

3 The Need to Prototype

The importance of prototyping as a risk reduction exercise within the software system development life cycle is becoming more widely accepted. Sommerville [6] and Pressman [7] both include sections in their books on software engineering, promoting prototyping as an integral part of the software production life cycle. Both authors highlight the benefits of prototyping as a verification and refinement exercise useful to augment the normal development approach but not to replace it. Indeed it may well be the case that the system/software requirements are well defined and fully understood from a previous implementation. In such a case prototyping would be both wasteful and unnecessary.

Mayhew and Dearnley's [5] prototype classificatior provides an appropriate framework within which to discuss the topic of prototyping in relation to the software system development life cycle.

3.1 Exploratory Prototyping

The early introduction of exploratory prototyping to assist in the derivation of a more complete and less erroneous requirements specification has significant effect in reducing the time and effort expended during the production phase of a project and also later during the maintenance phase. This in turn significantly reduces the overall cost of the project and in general produces a product that is more robust and far more acceptable to the customer. Boehm [8] highlights the effects of a badly stated requirement specification on the latter phases of the development life cycle and also during the in-service maintenance phase. He points out that the majority of program maintenance is not due to the correction of erroneous program code but modification to support changes to, or errors in, the original requirements specification.

Exploratory prototyping however, as stated by Gomaa [9], 'does not eliminate or reduce the need for a comprehensive analysis and specification of user requirement, it merely provides assistance in these activities'. Exploratory prototyping is particularly effective in situations where the customer cannot use past experience to specify what is required, which is usually the case when defining a radically new system. In such cases, past experience will be of only limited value in the specification of the new system.

The use of exploratory prototyping to establish the human computer interface requirements, in conjunction with the end user, is one important and widely accepted practice. This may take the form of a simple interactive prototype exhibiting superficial resemblance to the end product that allows preliminary definition/investigation of the cosmetics of the user interface or a sophisticated prototype providing a realistic and representative imitation. In highly user interactive systems, where efficiency, accuracy and ease of operation are necessary, it is vital that the requirement specification of the user interface be derived through practical interaction with a representative system. This is particularly important in the case of systems where an erroneous or delayed interaction could have severe consequences and in the worst case lead to loss of life. While it is possible to capture the total interaction via a paper model, as Weiser [2] indicates 'the use of a scale model allows the intended users of the system to react directly to the prototype as they would to the real system, rather than having to understand an esoteric design language. A user interface prototype will attempt to capture the idiosyncrasies of interactions with the proposed system.'

3.2 Experimental Prototyping

The use of experimental prototyping at the production stage is normally directed towards areas identified as 'high risk' during the risk analysis exercise. The focus of experimental prototyping is directed towards areas of the software system identified as particularly complex, highly dependable or performance critical.

The construction of a **design prototype** or prototypes, in order to investigate the technical merits of one or more possible solutions to an identified problem, allows the designer to gain a better insight into the area in question and to proceed with the design having confidence that the end result will provide an implementable and useable solution.

In systems where performance is a key issue, the construction of a **performance prototype** or prototypes enables investigation of time or resource critical areas of the design. This form of prototype necessitates use of the proposed development facilities and target execution environment in order to obtain relevant and meaningful results. This in itself allows practical evaluation of the development tools and provides a means for gaining familiarity with the target environment. In some cases it may be necessary to conduct a performance analysis exercise with the aid of a performance modelling tool in order to identify the critical areas of performance within the system. Results obtained through execution of a performance prototype may then be used to calibrate the theoretical model. Through the process of iteration, a stable model may be derived with which 'what if' question and answer sessions may be conducted in order to gain further insight into the performance related behaviour of the system.

Hardware prototypes are constructed in order to assist in the selection of system hardware. This may be necessary to assist in hardware/software trade-offs or simply to allow a choice to be made between components based on such factors as functionality offered, type of interface or simply ease of integration with the software.

3.3 Organisational Prototyping

Organisational prototyping is quite distinct from exploratory or experimental prototyping, in that it is used to address the issues of compatibility of the end product with the intended environment. The purpose of the prototype is not to assist in the derivation of a specification or design solution for a particular system but rather to investigate the completeness of the proposed solution in terms of both its ergonomic and functional characteristics.

Ergonomic prototypes are concerned with assessing the physical acceptability of the system in terms of the mechanics and efficiency of operation, aesthetic qualities, general health and safety considerations or simply correct size and shape for the target environment.

Functional prototypes are probably the most widely used and earliest form of prototyping undertaken. They concentrate on providing an implementation of the behaviour of the system, without necessarily using the final algorithm, implementation language or execution environment. Their prime objective is to provide a platform for evaluation of the completeness and correctness of interpretation of a requirement specification, with regard to what the system is required to do, ignoring issues of optimal design and performance.

3.4 The Benefits of Prototyping

The undoubted benefit of prototyping is the acquisition of additional knowledge about the system, gained by interaction with the environment within which it will eventually reside as a

product. This allows a more complete specification of the product to be produced. It allows the designer to prove the technical feasibility of his design and, equally important, that the functionality offered by his design matches the customer expectations. Prototyping reduces the risks associated with the production of complex, highly interactive and performant systems which in turn must reduce the overall cost. Brooks [10] suggests that developers should throw away the first implementation of a system, accepting that it will be wrong first time around. Prototyping of selected risk areas identified within a system is one way to counter this suggestion and ensure, as far as is possible, that the end product will be correct.

An often neglected benefit of prototyping is the visibility the customer gains of the production process. Prototypes provide a basis for customer dialogue and can serve as a platform through which to demonstrate the progress of a project with regard to overall appreciation of both the requirement and major technical issues of development. As Carey and Mason [11] point out 'with prototypes a distinct attempt is made to produce a "specification" which users can directly experience. Communication with users, particularly the non-specialist, is a major motivator behind prototypes'.

3.5 The Prototype as a Throw Away Item

Prototyping should be considered an integral part of the software system development life cycle. To be effective and to be widely accepted as a necessary process within the life cycle, prototyping must be comparatively inexpensive and provide relatively quick but positive results.

Most prototyping exercises that are undertaken are conducted as a separate and parallel exercise to the main stream development. In order to meet the stated objectives within a short time scale and with minimal cost it is necessary to relax the stringent quality standards applied to the development of the product. As a result, the developed prototype or prototypes are normally minimally documented, lacking in development history and due to the rapid and iterative nature of their development, possibly badly structured. The implementation language and operating system sometimes differ from the intended product and the items themselves are likely to lack the reliability and robustness required of an end product. In addition such issues as design for portability, reusability and maintainability may have been sacrificed for speed of development.

Given this general approach to prototyping, it is difficult to justify the inclusion of even the most useful components of the exercise into the main product development. In the absence of essential quality assurance techniques and without the ability to measure the quality of a software system component, independent of the process used to produce that component, it is difficult to accept the product of the prototyping exercise as being anything other than a throw away item Somerville [6] recommends that 'the prototype should be considered as a "throw away" system and should not be used as a basis for further development'. This is also consistent with Mayhew and Dearnley's [5] prototype classification which omits Law's [4] evolutionary class of prototype.

Davis [12] uses the acronym RAMP, reliability, adaptability, maintainability and performance, to capture the essential system properties which are generally ignored when prototyping and suggests that 'technology is not yet available to retrofit RAMP requirements'. He suggests that 'evolutionary prototypes will become more practical in the future as techniques for retrofitting RAMP requirements are developed'.

While it is desirable to completely integrate the prototyping process into the development life cycle, this is only achievable if the underlying life cycle model is flexible enough to accommodate prototyping and a suitable prototyping methodology is adopted, thereby ensuring adequate quality

assurance. Alternatively the chosen development environment should be rich in tools to assist or even automate prototype generation. Under such conditions the evolutionary prototype so far rejected will have a place and prototypes will become bi-products of the development life cycle rather than products of a less stringent and separate development process.

4 The Traditional Life Cycle Omits Prototyping

The conventional approach to software system development has been criticised due to its emphasis on a static paper representation of the specification or design, coupled with deferral of implementation until very late in the life cycle. Prototyping, as so far defined, lies outside of the scope of the traditional life cycle model and is often conducted as a separate parallel exercise, normally limited to exploratory prototyping of the human computer interface or organisational prototyping at the functional level. Effort is usually directed towards the prototyping of the areas which are difficult to specify using the conventional document driven development approach.

4.1 The Conventional Approach

The conventional software development life cycle model, more commonly known as the waterfall model, has provided the basis for software production for more than a decade. The waterfall model, defined as early as 1970 by Royce [13] and refined by Boehm [14] in 1976, provides a systematic approach to software development through a sequence of distinct development phases. The life cycle begins with a system requirements phase and progresses through the software requirements, preliminary design and detailed design phases deferring implementation detail until the latter code and systems test phases of the life cycle.

The waterfall model dictates a series of phases with verification, validation and feedback occurring at each phase, satisfactory completion of one phase being a prerequisite for progression to the next phase. The waterfall model enforces a document driven approach to software development requiring increasing elaboration of specification of each component, to a uniform level of detail, at each phase of the life cycle. The model provides project management with readily identifiable milestones, corresponding with completion of each life cycle phase, and allows the progress of the project to be measured according to the successful achievement of each milestone. The underlying assumption of the waterfall model, as pointed out by Agresti [15], is that 'successful software is developed by successively achieving sub-goals which correspond to the generation of intermediate products at each milestone'.

4.2 The Weakness of the Waterfall Model

The waterfall model has provided a much needed framework for software development for more than a decade but its usefulness is now in question. As highlighted by Agresti [15], the documentation driven approach reflects the needs of the period in which it was developed, a period lacking availability and access to cost effective hardware and software resources. This lack of resources and sophisticated tools to support rapid implementation and experimentation, coupled with the experiences of badly produced (hacked) systems, concentrated effort in providing a systematic and highly analytic development model. The model focuses on the analysis and design specification activities in recognition of the problems caused by premature entry into the coding process which was common practise at the time of its introduction.

The waterfall model relies on the domain knowledge of individuals combined with their interpretation of the perceived customer needs as a basis for the specification of each software component. In a situation where the product is new or the customer is unsure of the requirements, or even both, the waterfall model becomes inappropriate.

While emphasising the production of complete and validated paper specifications at each phase of the life cycle, the model completely ignores the maintenance of these specifications in the light of

errors discovered during the latter code, test and maintenance phases. Indeed the model assumes the validated specifications to be correct. Consequently users of the model tend not to maintain total consistency between the specifications and the delivered product due to the enormous cost in time and effort required to re-work the original and subsequent affected documentation.

4.3 The Need for an Alternative Model

The recent availability of high performance hardware and the emergence of sophisticated development tools brings into question the approach of deferral of implementation until late in the life cycle. Indeed the undoubted benefits of prototyping throughout the whole of the life cycle, as have been discussed, can now be realised. This questions the applicability of the waterfall model as a basis for all software system development and highlights the need for an alternative or extended model which accommodates prototyping as an integrated activity within the development life cycle. It should be noted that the author is not advocating the total rejection of the waterfall model but suggesting limiting its use to a category of well understood and familiar problems where the introduction of prototyping would be of little or no benefit.

While prototyping is becoming more widely accepted as being beneficial to the software system development process, it is still yet to be widely adopted in the development of real-time systems. This may be attributed to the large investment by the industry in the development of standards and procedures to support software system development using the conventional waterfall model. Alternatively, it may be that the industry finds it difficult to envisage the introduction of a development process where iteration and re-work are not only accepted as a natural consequence of the process but are encouraged by that process. It is understandable that the industry will view the process with a certain scepticism until a defined framework and methodology for prototyping not only exist but are proven in the field.

5 A Life Cycle that Incorporates Prototyping

The main objective of prototyping is to reduce the risks associated with the production of a system and in doing so raise the quality of the generated product. Given these objectives, it is difficult to argue against the inclusion of prototyping as an integral process within the software system engineering life cycle. It is ironic that the lack of quality control applied to the prototyping process itself, has restricted the benefits to be gained from prototyping. If prototyping is to be integrated within the development life cycle, it must be supported organisationally, methodologically and with adequate tool support not only to aid in the prototype production but also in the prototype assessment. Given such support the prototyping process will provide a valuable quality assurance technique and increase the benefits to be gained from prototyping by the possible re-use of prototype components in the production item. A candidate life cycle model which readily incorporates prototyping, with the flexibility to choose if and when prototyping is necessary, is the spiral model.

5.1 The Spiral Model as a Framework

Boehm [16] introduced the spiral model to provide a framework for guiding the software development process from a risk driven approach as opposed to the conventional documentation driven approach. The spiral model focuses on the formulation of strategies for resolving areas of risk. The model is flexible and accommodates any mixture of approaches including both specification oriented and prototype oriented. The spiral model allows the choice of a strategy appropriate to the particular development problem and risks. The spiral model, as its name suggests, involves a cyclic activity where progression within a cycle involves the same sequence of steps for each component of the product throughout each of its levels of elaboration. The spiral model is illustrated in Figure 3.

The spiral model allows for the progressive generation of specifications incorporating varying levels of detail, with the ability to defer the elaboration of low risk elements until the high risk elements have been prototyped, understood and specified. This risk driven approach allows the tailoring of the model towards an equivalent waterfall model, in the case of projects with low technical risk but critical budget, schedule and control.

While the spiral model provides an organisational framework for conducting the software system development process, into which prototyping may be integrated, it does not provide a methodology for prototyping. The lack of such a methodology to date may be yet another reason for the reluctance by the software system industry to introduce prototyping into the development life cycle. Strict management of the prototyping process is essential in order to sensibly constrain the process to operate within defined objectives, timescales and scope.

The prototyping exercise itself will identify the need for change both at the local component and system level, and management must have the ability to monitor change proposals, assess the implications of change and confirm any amendment to the appropriate level of specification. Mayhew, Worsley and Dearnley [17] propose a method for controlling prototype development, which they have named the 'Change Classification Method'. The method has been used on a commercial project in the UK with some success, although as the authors point out 'there remains a great deal of work to be carried out in the prototyping control area'.

5.2 The Change Classification Method

The change classification method addresses five issues highlighted by Mayhew, Worsley and Dearnley [17] as fundamental to assist the management of any project which incorporates prototyping. These they summarise as follows:

- To assess and control the effects of prototyping, to determine its impact on the project as a whole both in terms of resources and timescale.
- To ensure that the prototyping process is productive and providing useful insight into aspects of the system.
- To ensure that the prototyping process is converging, not contradicting previous prototyping results.
- To recognise and control the effects of change propagation which would otherwise disrupt aspects of the system thought to be complete.
- To control the actual prototyping exercise, organisationally.

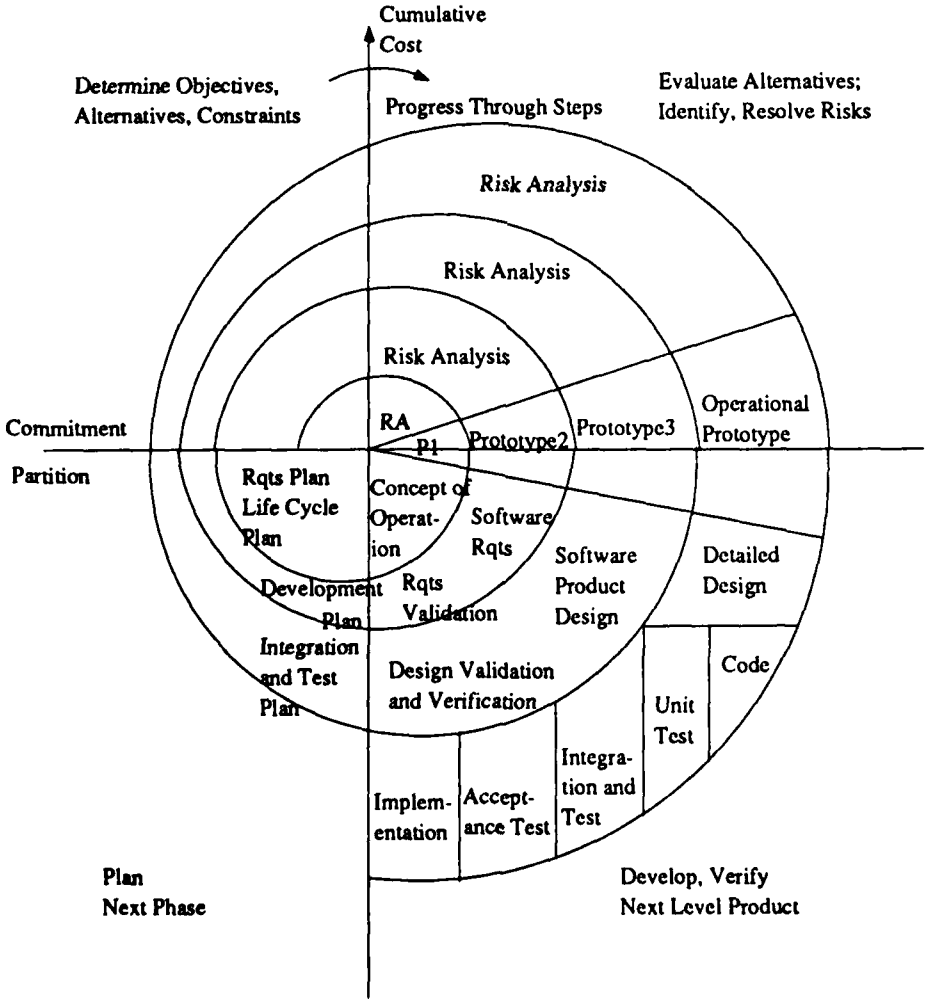


Figure 3: Boehm's Spiral Model.

The method assumes the objectives of each prototype iteration exercise to be well defined and appropriate milestones set, prior to the exercise beginning. The method centres around the monitoring and capturing of change requests, suggestions and comments generated during the prototype evaluation exercise. The types of change are categorised as one of either **cosmetic**, **local** or **global**.

- **Cosmetic** changes are considered trivial and have no repercussions elsewhere in the system. They are purely concerned with the formatting and presentation of already available information to the end user.
- **Local** changes are those which have a relatively local impact on the system but require additional information to be presented or new local facilities to be added. These highlight minimal local design changes.
- **Global** changes are those which have a significant impact on the rest of the system outside of the component being prototyped. These will highlight the need for substantial amendment to the proposed design.

The above classification is used to distinguish the scope of each suggested change and is used to indicate which set of procedures to follow in order to react to the suggested change. The method insists all change requests are recorded regardless of their class. The action procedures followed differ significantly depending on the class of change highlighted. Cosmetic changes are directly incorporated into the current prototype component. Local changes require that the current prototype is backed up prior to the changes being made to the prototype component. Each local change is also subject to a design inspection to validate the corrective action. Global changes require a design review and are not incorporated into the prototype. The implications of such changes require careful consideration and justification before being accepted as necessary. The review requires participation from both senior technical and managerial personnel to decide if and in what manner the changes will be made.

The change classification method uses the three categories of change request to provide management with a means of assessing the scope of the suggested changes. The accompanying change procedures provide the necessary controls for each iteration of the prototype. The record of all changes provides a valuable history of the progress of the prototype. The confirmation necessary for global changes allows project management to retain control and the regular design reviews and inspections necessary for all but cosmetic changes provides valuable and rapid feedback on the technical aspects of the prototype. The adoption of a set of procedures by which to control and monitor the prototyping process also overcomes certain of the quality issues previously identified with prototyping.

5.3 A Sensible Combination

The spiral model provides a suitable framework for the software system development life cycle with the risk driven approach encouraging prototyping where necessary. The change classification method for controlling the prototyping process provides a much needed basis from which to evolve the control aspects of prototyping. As a combination they provide a sensible basis for the software system development life cycle, which is likely to provide the customer with the product he needs rather than the one he thinks he needs. The element of control introduced by the change classification method should allay the fears which have so often led to the rejection of prototyping as an integral component of the software system development life cycle. With wider adoption of the spiral development model being inevitable as customers insist on phased deliveries of large

systems, full advantage of the ability to prototype will be taken and the need for a formal approach to prototyping will become necessary.

The addition of a suitable analysis tool, capable of assessing the quality of the software within the prototype implementation and assisting the identification of possible re-usable components for inclusion in the production item, would greatly enhance the benefits to be gained by adopting a prototyping approach. The combination of such a facility and the prototyping methodology discussed, would provide the necessary quality assurance so far lacking in the prototyping environment and remove a major obstacle which has inhibited the effective re-use of prototype implementation. The notion of prototypes being 'throw away' could then be reviewed.

The economic benefits to be gained from the re-use of suitable software components will encourage the adoption of a prototyping approach, but these immediate short term benefits should not obscure the overall quality enhancement and associated long term benefits to be gained from prototyping, even in the current 'throw away' form.

6 Prototype Implementation Approaches

When considering prototype implementation approaches, the intent of the prototype itself is the key factor in deciding which particular approach is appropriate for a given situation. It is therefore sensible to discuss the topic using Mayhew and Dearnley's [5] prototype classification as previously discussed.

6.1 Implementing Exploratory Prototypes

Exploratory prototypes assist the iterative process of requirements elicitation and verification, by providing working models through which dialogue developers, systems analysts and prospective users may directly observe and experience the system behaviour. Working models not only provide the platform through which the human computer interface may be developed and refined, but also assist in the overall systems analysis exercise, by conveying the results of the analysis to the end user in a non technical form. The prototype iteration may serve as a learning process for all involved, particularly in cases where the initial requirements are vague. A major benefit to be gained from exploratory prototyping is the removal of the communications barriers which often exist between the end users and the system developers. An executing model provides the platform through which the end user can articulate his needs without having to understand specialised design notation.

Exploratory prototyping is primarily concerned with the derivation of the human computer interface which enables the end user to effectively carry out his system tasks in close cooperation with the underlying machine. In order to easily develop interactive exploratory prototypes, it is necessary to view the human computer interface as an integral but clearly delineated part of the system. Separation of the system dialogue from the system computation allows decisions affecting the dialogue to be completely isolated from the rest of the system. Hartson and Hix [18] provide an excellent paper focusing on the theories, the methodologies and the tools for incorporating dialogue design principles into human computer interfaces.

The concept of dialogue independence is fundamental to the success of exploratory prototyping. The concept allows the prototyper to concentrate on the end user issues, ignoring the computational detail of the underlying application. With the assistance of development tools which support the concept, exploratory prototyping can be both cost effective and extremely expressive. Hartson and Hix [18] provide a comprehensive summary of both research and commercially available tools to assist human computer interface prototyping.

Current development tools available to assist exploratory prototyping fall into one of the following categories:

- User Interface Management Systems (UIMS).
- User Interface Toolkits.
- Object Oriented Languages.

The tools, which fit into one of the above categories, all have the same objective but their approach is distinctly different. Bass and Coutaz [19] provide a useful overview of the different approaches and Dickenson, Benton and Atyeo [20] provide a brief summary of useful prototyping tools. A comprehensive summary of commercially available tools with techniques to assist human

computer interface design and rapid prototyping, is presented by Overmyer [21]. The above categories are ordered to reflect the level of abstraction of the implementation offered, UIMS offering very high levels of abstraction and object oriented languages very low levels. The degree of dialogue independence and the level of interface offered by the tools themselves are major issues. These determine the degree of programmer assistance necessary when exploratory prototyping.

A novel approach to exploratory prototyping which utilises the higher level tools, is storyboarding. Storyboards are interactive models depicting the system functionality constructed from a sequence of pre-defined display screens linked to input dialogue selections. The screens may be raster images generated from scanned pictures or the raster output of graphics tools. Storyboarding has proven to be a relatively cost effective way of exploratory prototyping. The approach is generally implemented using UIMS or User Interface Toolkits which execute on low cost hardware platforms offering reasonably high resolution screens. Andriole [22] describes his experience of storyboarding as 'a viable alternative to conventional modeling and prototyping'. The technique has proved to be extremely successful in assisting with the derivation of the specification for the Software Measurement and Analysis Testbed (SMAT), an environment being developed within the Software Engineering Group of Information Technology Division.

6.1.1 User Interface Management Systems (UIMS)

A User Interface Management System (UIMS) is an environment which supports the specification and execution of a human computer interface, in a dialogue independent and programming language independent fashion. The dialogue developer may define the interface through a declarative definition language using a standard text editor or by direct interaction with the UIMS. The UIMS translates the dialogue description, into an executable representation and produces the interface at run-time. A key feature of a UIMS is the ability to execute the dialogue without an application system being present.

UIMS provide a very high level of interface to the dialogue developer along with facilities to easily and quickly change the definition of the interface, without the need for programming support. UIMS are normally built upon a user interface toolkit, which provides the workstation and dialogue management. The dialogue independence offered by UIMS makes them ideal for exploratory prototyping.

Freeman [23] provides an overview and assessment of Open Dialogue, a UIMS produced by Apollo, from work carried out at the Rutherford Appleton Laboratory. The paper summarises the capabilities of this powerful UIMS which is commercially available for UNIX platforms. Bass , Hardy et al [24] and [25] provide two technical reports which describe the work carried out at the Software Engineering Institute, Carnegie-Mellon University, describing their research UIMS, Software Engineering Rapid Prototyping Environment (SERPENT). Both UIMS offer a high level of user interface through declarative languages, are dialogue independent and are built upon the X Windows System [26], a windowing library/toolkit developed at the Massachusetts Institute of Technology (MIT).

An alternative and higher level of dialogue definition is provided by UIMS which offer an interactive dialogue definition interface. Examples of interactive dialogue definition UIMS are Prototyper, a Macintosh specific UIMS built upon the Macintosh Toolbox, and The Object Oriented Graphical Modeling System, produced by Sherrill-Lubinski, which can be ported to numerous vendors platforms (Sun, DEC , Silicon Graphics). The latter is built upon the hardware vendors proprietary toolkit (e.g. SunView, DECwindows) and can be considered a UIMS plus, in that it also

provides facilities for animation of the interface via a user defined database. Although not designed explicitly for this purpose, Hypercard, the Macintosh information retrieval and presentation system, may also be considered as an UIMS. The facilities provided are more than comparable with those of purpose built interactive dialogue definition UIMS.

6.1.2 User Interface Toolkits

A User Interface Toolkit is a library of routines and an underlying run-time kernel which provide a high level interface through which human computer interfaces may be developed at the programming level. The toolkit provides facilities which allow the programmer to define the cosmetics of the interface and the routines which are to be invoked under defined dialogue conditions. Typical facilities range from workstation management such as windowing, graphics and text editing to dialogue handling in the form of menus, buttons and panels selected by mouse interaction.

The emergence of high powered graphic workstations has seen such toolkits arrive as standard items supplied with the operating system, but unfortunately in the form of proprietary software. With standardisation and open system computing being key issues, the next generation of toolkits will cater for greater portability. The X Windows System [26] is emerging as the industry standard and will allow not only portability of developed software but also interoperability with different vendors equipment.

While such toolkits offer a great deal of flexibility in generating the interface, they do not enforce the distinction between user interface and application and they also require considerable programming effort to manufacture an executable interface. User Interface Toolkits however provide the infrastructure on which many UIMS have been developed, and due to their lower level of interface offer a means of overcoming problems often encountered at the higher level.

6.1.3 Object Oriented Languages

Object oriented languages provide many of the features readily applicable to exploratory prototyping, through inheritance, dynamic binding and abstraction. In addition, the language environments include as fully integrated, such facilities as windowing and dialogue interaction.

Smalltalk is a programming language which after various incarnations has emerged in two major commercially available forms, Smalltalk-80 and Smalltalk/V. Smalltalk was developed in 1970 at Xerox's Palo Alto Research Centre. A Smalltalk program is made up of Objects, Methods (operations offered by objects), Messages (which allow objects to communicate) and Classes (from which objects inherit the same structure and operations). Smalltalk-80 executes on a Sun UNIX platform using SunTools. Smalltalk/V, a product of Digitalk Incorporated, is an example of a low cost environment which executes on IBM PC or compatibles and Macintosh platforms.

Although at first sight these languages offer a great deal more than conventional procedural languages, they are often difficult to learn and, due to their run-time binding, perform relatively slowly. Exploratory prototyping using object oriented languages would require full time programming assistance and is therefore less attractive.

6.2 Implementing Experimental Prototypes

Experimental prototypes assist the software system designer in the derivation of solutions to problems related to specific areas of the software design. They allow the designer to determine

the feasibility and acceptability of solutions to a particular problem. Experimental prototyping is divided into three sub-classes as previously described. Implementation approaches for each of the three sub-classes are described below.

6.2.1 Design Prototypes

Design prototypes, as the name suggests, assist the investigation of design issues. The terminology is used explicitly in the area of software design and within this context covers two distinct types of prototyping:

- Prototyping in the Large.
- Prototyping in the Small.

Prototyping in the large refers to the experimental development of a skeletal software architecture, which will eventually provide the basis for the development of the complete system software. This may be necessary in order to define the interface to external users of the system/software, to assist in the definition of sub-systems and their interfaces or simply to assist the problem of partitioning and constructing large systems. Prototyping in the large becomes particularly relevant when building large real-time distributed systems. The Ada programming language provides suitable constructs which allow compilation and construction of a system with minimal effort from its interface specifications and as such lends itself to supporting the generation of such a framework.

Prototyping in the small refers to the experimental development of specific elements of the software in order to investigate detailed design issues, evaluate alternative algorithms or simply to assist in the derivation of a solution to a new and difficult problem.

Both of the above types of design prototyping would normally be conducted using the target implementation language, but not necessarily using the target hardware environment. A combined approach, utilising both prototyping in the large and prototyping in the small, provides the mechanism for addressing risk, both in terms of the breadth and the depth of the software design.

6.2.2 Performance Prototypes

Performance prototyping refers to the experimental development of combined and representative hardware/software elements of the system, in order to investigate critical timing, space or throughput requirements. Investigation of performance at the component level is best achieved using non-intrusive monitoring tools. Hewlett Packard produce a series of 'intelligent' logic analysers with both hardware and software options to assist performance analysis, and Cadre Technologies offer their Software Analysis Workstation (SAW) which provides an integrated set of hardware/software facilities to measure and verify performance. While it is not possible to guarantee the total system performance from the results of such specific prototypes, a combination of prototyping and systems performance modelling can provide valuable insight into overall system behaviour.

The Performance Analyst's Workbench System (PAWS) [27], a product of Information Research Associates, provides the ability to model software, hardware and human components at any desired level of detail, based on a modelling methodology using pictorial representation and a simulation language to describe and evaluate the pictures.

Such modelling tools allow the system behaviour to be captured, and by simulation, provide estimates of the system performance. Calibration of the models, using practical results obtained from specific executing components, allows evolution towards a stable model offering a greater degree of confidence of its output results. The models are particularly useful in indicating the sensitivity of particular system parameters and as such assist in the risk assessment exercise by indicating areas requiring further performance prototyping.

6.2.3 Hardware Prototypes

Hardware prototypes are primarily used to assist in the selection of system hardware, but they also provide the mechanism for investigation and familiarisation with detailed software/hardware integration issues. The construction of representative and inter-connected physical processing elements of a system allows sustained operation of the chosen hardware in the target situation, in order to verify environmental and reliability requirements. In addition, the integration of new components with existing hardware and software elements of the system, allows compatibility checks of both the physical interfaces and software/hardware protocols. The construction of hardware prototypes allows both hardware and software engineers to fully understand the low level interfacing detail early in the development, thereby allowing an easier transition of production software from the host to the target environment.

6.3 Implementing Organisational Prototypes

Organisational prototyping involves the end users of the system operating the prototype in the target environment. This gives the user the ability to assess its functional completeness and overall suitability, in addition to any organisational issues which may arise due to the introduction of the new system. Organisational prototyping is divided into two sub-classes as previously described. Implementation approaches for each of the two sub-classes are described below.

6.3.1 Ergonomic Prototypes

Ergonomic prototyping refers to the generation of a 'mock-up' or 'scale-model' of the physical system with which the human interacts. Ergonomic prototypes allow evaluation of mechanical issues, such as correct position and posture for ease of reach of controls, effects of physical layout on human efficiency of operation, and aesthetic acceptability to the end user.

Ergonomic prototypes are often built to scale from materials that are cheap to use and quick to sculpture into the correct physical shape. Elements of the prototype may however be final target system components, particularly in cases where it would be difficult and non economical to substitute for them, or where their replacement with a mock-up would detract from the overall effect of the prototype.

6.3.2 Functional Prototypes

Functional prototypes create an illusion of the final system/software, in terms of the functionality and data transformations offered, by mimicking the real system algorithms. The implementation of functional prototypes may be accomplished using alternative languages and execution environments to the target system, as optimisation of performance or accuracy of results is not an issue. Functional prototypes may be implemented at one of two levels, system or software.

System functional prototypes exhibit the behaviour expected by external interfaces of the system, whether human or other systems. In the case of human interfaces, functional prototyping is

very similar to exploratory prototyping, except the intent is very different. Rather than eliciting the requirements, the prototype provides a vehicle to demonstrate the completeness and/or correctness of the systems functionality, and its compatibility within the environment in which it will operate. Many of the techniques discussed for exploratory prototyping are applicable for system functional prototyping involving human users, in particular storyboarding. In the case of non-human system interfaces, the prototype merely emulates the data and controls required by the connected system.

Software functional prototypes are smaller functional component of a system, exhibiting representative functionality in order to confirm the completeness and/or the correct behaviour of the component when integrated with the rest of the system. The functional prototype may use non representative input and output internally, and mimick the behaviour expected by other software components of the system. Software functional prototypes are usually more restricted in terms of the chosen implementation language, due to the problems of interfacing to other existing target software components.

7 Future Trends in Prototyping

The advent of Computer Assisted System/Software Engineering (CASE) has already had a great impact on the software system production industry. Many companies now employ many and various CASE tools to automate the software system development life cycle. As the CASE industry matures and vendors tools become more integrated and more sophisticated, the ability to automatically generate prototypes throughout all stages of the life cycle will be possible.

Already tools exist to animate and prototype at the requirements specification level. Statemate from i-Logix Incorporated[28], provides facilities to capture and analyse a system specification, based on Harel's [29] statecharts, with facilities to carry out animated execution and simulation of the described system. In addition Statemate provides an automatic translation of the requirements specification into the Ada language, which provides for prototyping at the specification level.

Many CASE tools are available that allow graphical representation of the system design with the ability to generate the skeleton code for the system. AdaGraph which uses the Pamela notation and Cadre Technologies Teamwork/Ada which uses Buhr notation are examples of commercially available tools. The Carleton Embedded Design Environment (CAEDE)[30], produced as a research project at Carleton University, provides the generation of skeleton Ada code from Buhr notation, plus facilities to analyse the design for deadlock and race conditions.

While many tools are available today, each tool covers a small part of the development life cycle and is normally stand-alone. The tool manufacturers have identified the need to support prototyping but do not address the life cycle as a whole. Consequently the use of these tools offers no means of traceability between each phase of the life cycle that each tool supports. Given the desire for prototypes to become bi-products of an evolutionary system development process, an integrated toolset is required which covers all phases of the life cycle and is capable of automatic generation of prototype systems. Individual tools make evolutionary development very difficult. While they automate the production of a prototype from a graphical notation at a particular phase of the life cycle, a great deal of manual effort is necessary to maintain consistency of descriptions across tools when using an iterative evolutionary approach. The lack of communication and therefore traceability between the different tools would make the process very error prone.

With the emerging standardisation of CASE data interchange formats allowing the possibility for totally integrated toolsets, and the realisation of the importance of prototyping as an integral part of the software system development life cycle, the future will certainly bring a host of tools which will provide for the automated production of evolutionary prototypes.

THIS PAGE INTENTIONALLY LEFT BLANK

8 Discussion

A survey of the literature on the subject of prototyping has revealed a multitude of papers discussing the topic. The interpretation and use of the terminology is in general extremely diverse and while a great number of papers extol the virtues of prototyping, few offer practical experience with the technique for the development of real-time systems. In addition little attention is given to the problem of incompatibility of prototyping and the traditional software system development life cycle. In general, the term 'prototype' is used to cover a wide range of definitions, with the most frequent use of the terminology occurring in the phrase 'throw away prototype' and 'evolutionary prototype' or derivations of these phrases.

The term 'throw away prototype' is repeatedly used to describe the model produced to assist requirements analysis and human computer interface capture. On a few occasions a 'throw away prototype' is defined as such, due to its relatively small cost of development compared to the overall system cost (the guide usually being approximately 10%). References to the terminology are repeatedly used to discuss prototyping early in the software system development life cycle but little reference is given to the technique being used to assist other phases of the life cycle where it is equally appropriate.

The term 'evolutionary prototype' is often wrongly used and interchanged with the term 'incremental development'. Davis [12] provides a definition of both evolutionary prototyping and incremental development which highlights the distinct difference in their meaning. The former implies a lack of knowledge of the requirement and the need for experimentation with an operational system in order to learn, whereas the latter implies implementation of a known requirement in subsets of increasing capability. The advocates of 'evolutionary prototyping' generally offer little or no advice regarding the quality assurance aspects of the software system being developed, neither do they propose a life cycle model or method for controlling the process. The technique is often proposed without any suggestion as to how key system properties, which may have been sacrificed to speed up the generation of the prototypes, may be later incorporated.

This paper presents prototyping as a necessary technique that has applicability throughout many phases of the real-time software system development life cycle. The key texts upon which the paper is based were chosen because of their coverage of the software system life cycle as a whole and their practical applicability to the production environment. While many papers were found which addressed the subject of prototyping, few discussed the subject in this wider context. A great deal of research has been directed towards front end prototyping, to assist requirements analysis and requirements specification, but very little attention has been given to the role of prototyping throughout the remainder of the life cycle.

Given that the necessary infra-structure to support a prototyping approach is now available, a commitment is required from both customer and supplier in acknowledging the significant advantages to be gained from adopting such an approach. The US congressional sub-committee's staff report, 'Bugs In The Program'[31], published in September 1989, recommends a new basis for software system procurement decisions. One of the key recommendations of the report is the adoption of both a prototyping approach and the use of an extended spiral model for software system development. This is in recognition of the fact that the document driven, specify-then-build approach, has caused so many of the US Defense Department's software problems.

THIS PAGE INTENTIONALLY LEFT BLANK

9 Conclusion

Prototyping is a valuable technique which can be used to augment the software system development life cycle and assist in the production of higher quality systems. Adoption of the technique requires a development life cycle model capable of supporting the iterative nature of the prototyping process and a method for managing the exercise.

The current state of the art, with respect to software system development environments, restricts prototyping to the category defined as 'throw away'. Throw away prototypes have applicability throughout many phases of the software system development life cycle. If throw away prototyping is to be seriously adopted and integrated into the software system development life cycle, the process must be fully defined and understood. The classification proposed by Mayhew and Dearnley [5], upon which this paper is based, is recommended as the basis for the definition of throw away prototypes.

Prototyping requires a flexible and supportive software system development life cycle model, and in order to attract both management and quality organisation approval, the process must have a defined method. The combined spiral model [16] and change classification method [17] described in this paper provide the necessary framework and controls to satisfy both the management and quality issues. Within this framework and with these controls, throw away prototyping can become an integral component of the software system development life cycle.

Although evolutionary prototyping is not yet possible with current technology, a hybrid combination of 'throw away' prototyping' and 'incremental development' used within the framework of the spiral model, offers an approach which is far superior to the traditional waterfall model, where applicable. The waterfall model still offers an adequate approach for the development of systems which are fully understood, and therefore by definition, require no prototyping.

THIS PAGE INTENTIONALLY LEFT BLANK

References

[1] Gregory S.T., "On Prototypes vs. Mockups", *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 5, p. 13, 1984.

[2] Weiser M., "Scale Models and Rapid Prototyping", *ACM SIGSOFT Software Engineering Notes*, vol. 7, no. 5, pp. 181–185, 1982.

[3] Floyd C., "A Systematic Look at Prototyping", in *Approaches to Prototyping* (Budde R., Kuhlenkamp K. et al, ed.), Springer-Verlag, 1984.

[4] Law D., "Prototyping: A State of the Art Report", tech. rep., NCC, 1985.

[5] Mayhew P.J. and Dearnley P.A., "An Alternative Prototype Classification", *The Computer Journal*, vol. 30, no. 6, pp. 481–484, 1987.

[6] Sommerville I., *Software Engineering, Third Edition*, ch. Requirements Validation and Prototyping. Addison-Wesley, 1989.

[7] Pressman R.S., *Software Engineering, A Practitioner's Approach, Second Edition*, ch. Software Prototyping. McGraw-Hill, 1987.

[8] Boehm B.W., "Some Steps Towards Formal and Automated Aids to Software Requirements Analysis and Design", *IFIP*, 1974. Amsterdam: North-Holland.

[9] Gomaa H., "The Impact of Rapid Prototyping on Specifying User Requirements", *ACM SIGSOFT Software Engineering Notes*, vol. 8, no. 2, pp. 17–28, 1983.

[10] Brooks F.P., *The Mythical Man-Month*. Addison-Wesley, 1975.

[11] Carey T.T. and Mason R.E.A., "Information Systems Prototyping: Techniques, Tools and Methodologies", *INFOR - The Canadian Journal of Operational Research and Information Processing*, vol. 21, no. 3, 1983.

[12] Davis A.M., "A Strategy for Comparing Alternative Software Development Life Cycle Models", *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1453–1461, 1988.

[13] Royce W.W., "Managing the Development of Large Software Systems: Concepts and Techniques", in *Proceedings of WESCON*, 1976.

[14] Boehm B.W., "Software Engineering", *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1226–1241, 1976.

[15] Agresti W.W., *New Paradigms for Software Development*. IEEE Computer Society Press/North-Holland, 1986.

[16] Boehm B.W., "A Spiral Model of Software Development and Enhancement", *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 8, pp. 14–24, 1986.

[17] Mayhew P.J., Worsley C.J. and Dearnley P.A., "Control of Software Prototyping Process: Change Classification Approach", *Information and Software Technology*, vol. 31, no. 2, pp. 59–66, 1989.

[18] Hartson H.R. and Hix D., "Human-Computer Interface Development: Concepts and Systems for its Management", *ACM Computing Surveys*, vol. 21, no. 1, 1984.

[19] Bass L. and Coutaz J., "Human-Machine Interaction, Considerations for Interactive Software", Tech. Rep. CMU/SEI-89-TR-4, Software Engineering Institute, Carnegie-Mellon University, 1989.

[20] Dickerson K.R., Benton R.H. and Atyeo M.A., "Rapid Prototyping Tools and Techniques", *British Telecom Technical Journal*, vol. 6, no. 4, pp. 65–68, 1988.

[21] Overmyer S.P., "Survey of Rapid Prototyping Tools for User-Computer Interface Design", Tech. Rep. CTC-TN-89-001, Contel Technology Centre, Contel Corporation, 1989.

[22] Andriole S.J., "Storyboard Prototyping for Requirements Verification", *Large Scale Systems*, vol. 12, pp. 231–247, 1987.

[23] Freeman T.G., "Experience with Open Dialogue", in *Proceedings of the Fourth Australian Software Engineering Conference*, pp. 11–16, 1989.

[24] Bass L., Hardy E. et al, "Introduction to the Serpent User Interface Management System", Tech. Rep. CMU/SEI-88-TR-5, Software Engineering Institute, Carnegie-Mellon University, 1988.

[25] Bass L., Hardy E et al, "The Serpent Runtime Architecture and Dialogue Model", Tech. Rep. CMU/SEI-88-TR-6, Software Engineering Institute, Carnegie-Mellon University, 1988.

[26] Scheifler R.W. and Gettys J., "The X Window System", *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79–109, 1986.

[27] Information Research Associates, *Performance Analyst's Workbench System, Introduction and Technical Summary*, November 1986.

[28] Harel D., Lachover H. et al, "Statemate: A Working Environment for the Development of Complex Reactive Systems", in *Proceedings of the Tenth International Conference on Software Engineering*, pp. 396–406, 1988.

[29] Harel D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[30] Buhr R.J.A, Karam G.M. et al, "Software CAD: A Revolutionary Approach", *IEEE Transactions on Software Engineering*, vol. 15, no. 3, pp. 235–249, 1989.

[31] Subcommittee on Investigations and Oversight, Committee On Science, Space, and Technology, U.S. House of Representatives, "Bugs In The Program", September 1989.

DISTRIBUTION
DEPARTMENT OF DEFENCE

	COPY NO.
Defence Science and Technology Organisation	
Chief Defence Scientist)	
First Assistant Secretary Science Policy)	1
Director General Science and Technology Programs)	
Counsellor, Defence Science, London	Ctr Sheet Only
Counsellor, Defence Science, Washington	Ctr Sheet Only
Defence Science Representative, Bangkok	Ctr Sheet Only
Scientific Adviser, Defence Research Centre, Kuala Lumpur	Ctr Sheet Only
Electronics Research Laboratory	
Director, Electronics Research Laboratory	2
Chief, Information Technology Division	3
Chief, Communications Division	4
Chief, Electronic Warfare Division	5
Research Leader, Information Systems	6
Research Leader, Information Processing	7
Head, Software Engineering Group	8 - 9
Head, Information Systems Research Group	10
Head, Trusted Computer Systems Group	11
Publicity and Component Support Officer, Information Technology Division	12
Aeronautical Research Laboratory	
Director, Aeronautical Research Laboratory	13
Chief, Aircraft Systems Division	14
Materials Research Laboratory	
Director, Materials Research Laboratory	15
Surveillance Research Laboratory	
Director, Surveillance Research Laboratory	16
Weapons Systems Research Laboratory	
Director, Weapons Systems Research Laboratory	17
Head, Combat Systems Integration Group	18
Navy Office	
Naval Scientific Adviser	Ctr Sheet Only
Army Office	
Scientific Adviser - Army	19
Director General Army Development (NSO), Russell Offices for ABCA Standardisation Officers	
UK ABCA Representative	20
US ABCA Representative	21
Canada ABCA Representative	22
New Zealand ABCA Representative	23
Air Office	
Air Force Scientific Adviser	24
Joint Intelligence Organisation (DSTI)	25

DOCUMENT CONTROL DATA SHEET

Security classification of this page : UNCLASSIFIED

1 DOCUMENT NUMBERS

AR
Number : AR-006-430

Series
Number : ERL-0526-TR

Other
Numbers :

2 SECURITY CLASSIFICATION

a. Complete Document : Unclassified

b. Title in Isolation : Unclassified

c. Summary in Isolation : Unclassified

3 DOWNGRADING / DELIMITING INSTRUCTIONS

4 TITLE

RAPID PROTOTYPING : FOR WANT OF BETTER WORDS

5 PERSONAL AUTHOR (S)

Stephen P. Jones

6 DOCUMENT DATE

September 1990

7 7.1 TOTAL NUMBER OF PAGES 32

7.2 NUMBER OF REFERENCES 31

8 8.1 CORPORATE AUTHOR (S)

Electronics Research Laboratory

8.2 DOCUMENT SERIES and NUMBER
Technical Report
0526

9 REFERENCE NUMBERS

a. Task : DST 89/047

b. Sponsoring Agency :

10 COST CODE

223 AA 254

11 IMPRINT (Publishing organisation)

Defence Science and Technology Organisation Salisbury

12 COMPUTER PROGRAM (S)
(Title (s) and language (s))

13 RELEASE LIMITATIONS (of the document)

Approved for Public Release

Security classification of this page : UNCLASSIFIED

Security classification of this page : UNCLASSIFIED

14 ANNOUNCEMENT LIMITATIONS (of the information on these pages)

No Limitation

15 DESCRIPTORS		**16** COSATI CODES
a. EJC Thesaurus Terms	Rapid Prototyping Software Engineering Systems Engineering	1205
b. Non - Thesaurus Terms		

17 SUMMARY OR ABSTRACT
(if this is security classified, the announcement of this report will be similarly classified)

Rapid prototyping is widely promoted as being an effective technique to assist the development of complex systems. While the phrase is commonly used and discussed within the software system development community, the interpretation is somewhat diverse and the technique itself has not been widely adopted in practice. This paper reviews the recent literature on the subject of 'rapid prototyping' and in highlighting key texts, aims to provide a definition of the terminology and tries to put prototyping into context as a necessary and integral component of the software system development life cycle. The paper discusses the incompatibility of prototyping and the traditional life cycle model and proposes an alternative life cycle model which incorporates prototyping as an integral activity.